

这个星期主要是实现了并行绘制。

首先定位了原来并行框架效率低的问题，发现是生成体数据时文件操作不正确占用了大量运行时间，而并行框架效率是非常高的（原来运行 32MB 体数据加一操作占用 22 秒，实际 21 秒用于生成体数据文件操作）。

然后测试了一下内存溢出问题。生成 1G（ 4^3 个 256^3 的体数据）产生内存溢出问题。后来编译依赖库 boost 为 x64 平台，将整个框架在 x64 平台下编译，运行没有问题。

然而系统中的 x64 OpenGL 除了问题，DLL 不匹配，造成不可预测的内存溢出问题，费了一天除错。于是为了节省时间，将绘制框架转回 x86 平台，以后再来解决这个问题。

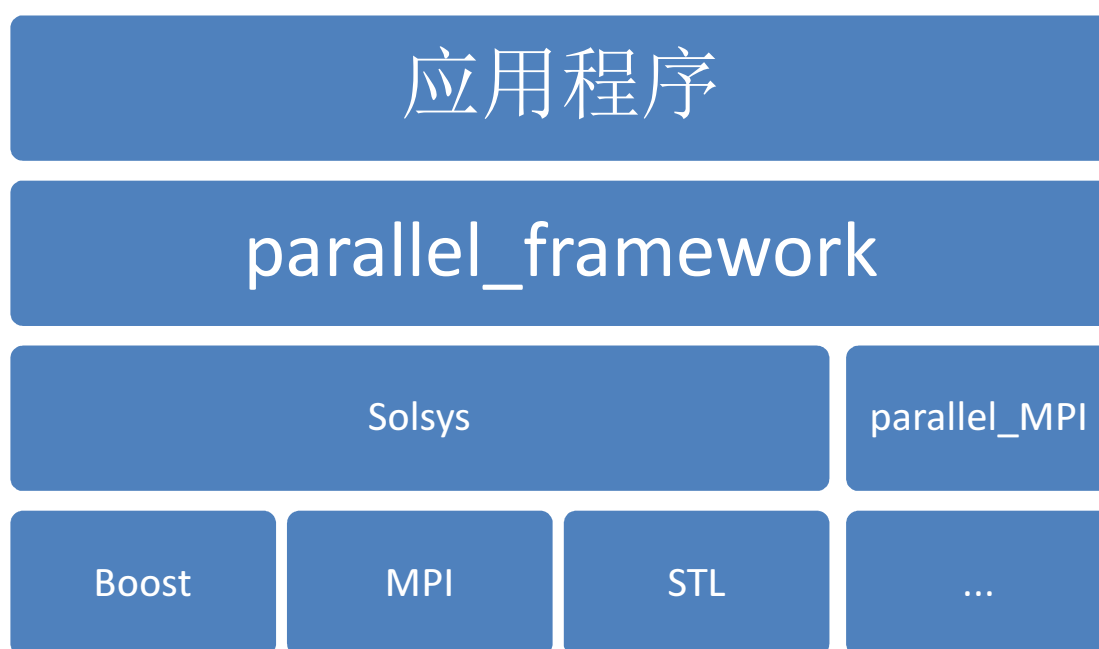
接着整理了原有并行框架，将框架分成 4 层（原来是 3 层）：

底层是 MPI、Boost 等常用库。

Solsys 是框架库，里面定义了各个接口以及调用。

parallel_framework 则是并行框架的特定实现，比如现有两个实现：单进程多线程并行和 MPI 并行，分别用于单机多核环境开发测试和小集群的运行。这一层是原来没有的。

最上层是应用程序，他会调用 parallel_framework，并实现各个 Operation。整个并行框架就是在各种 Operation 的组合下实现不同的绘制功能。



最后是真正实现了并行绘制。

主要是做了几个工作：

编写了 Operation_Render。这个 Operation 实现了 Graphics（前三个星期写的基于 OpenGL 的图形库）的初始化、资源创建和绘制操作。其中遇到了多 OpenGL Context 运行于多线程，在多窗口绘制的问题，网上根本没有任何资料。试验了解到，为了在多个窗口绘制，必须创建多个 OpenGL Context（单个窗口可以由多个 OpenGL Context 绘制，它们不是一对一关系），然后可以通过 glMakeCurrent 函数将 OpenGL Context 绑定到执行 glMakeCurrent 的线程，每个 OpenGL Context 同时只能被绑定到一个线程上，每个线程也只能绑定一个 OpenGL Context，

所以它们是一对一的关系。而线程之间是不会相互影响的。从而实现在同一个进程中的并行绘制。事实上，在显卡驱动程序中，四个 OpenGL Context 产生的 OpenGL Command 最终还是会被串行执行。

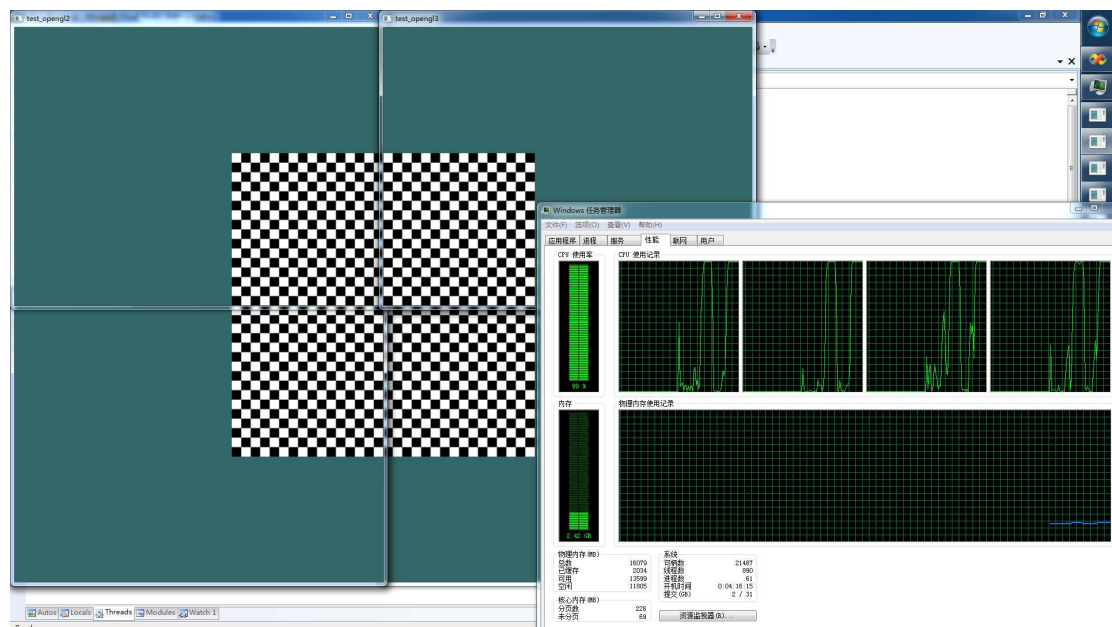
其次是试验了生成 Frustum 投影矩阵的算法。因为原来是 Perspective 投影矩阵，而要做大屏幕分割，必须将原来的 Perspective 投影矩阵分割为若干个 Frustum 投影矩阵。在两星期前编写 CPU 版图形库时学到，Perspective 投影矩阵实际上是 Frustum 投影矩阵的(0,0,0)在中间的特殊情况。所以现在做这个试验，并实现分割的算法问题也是迎刃而解。

代码如下：

```
void CameraUtility::GetPerspectiveMatrix( float44 * const matrix , FLOAT const pfovy , FLOAT const paspect ,
FLOAT const pznear , FLOAT const pzfar )
{
    FLOAT const ymax = pznear * tanf( pfovy / 2 );
    FLOAT const xmax = ymax * paspect;
    GetFrustumMatrix( matrix , -xmax , xmax , -ymax , ymax , pznear , pzfar );
}

void CameraUtility::GetFrustumMatrix( float44 * const matrix , FLOAT const pleft , FLOAT const pright , FLOAT
const pbottom , FLOAT const ptop , FLOAT const pznear , FLOAT const pzfar )
{
    ...
}
```

具体可参阅 Solsys 中的代码。



可以看到，四个线程占用了两个核心和两个 HT 技术虚拟出来的核心，完成并行绘制。风别绘制在四个窗口中。CPU 占用率达到 100%（如果是单线程，或者是串行的，占用率只能到 25%）。现在运行于 Windows，可以直接在 Linux 下编译运行，因为底层库已经在前两个月的努力中实现统一的接口。等 MPI 的消息功能实现后，就可以做小集群的并行绘制测试。

下星期首先要想办法将完整的地球纹理下载下来。然后采用基于 **Image Space** 分割并行绘制地球（这里不基于 **Object Space** 是因为网格数据不像体数据，数据量比较少，每个节点都可以存储整个网格数据，比较适合 **Image Space** 的分割方法）。